A Session Programming Tutorial for **SessionJ**

Raymond Hu

August 19, 2008

Contents

1 Introduction

This note is a tutorial on writing, compiling and running **SJ** programs. The aim is to simply cover the basics of session programming in **SJ** from a practical perspective; a more detailed specification of the **SJ** language is presented in [7]. Note that *session*, in the context of this work, means a *binary* session, i.e. an interaction between two parties.

The latest version of **SJ** can be downloaded from the project homepage [8]. This page also links to further information, such as papers [4] and benchmark results. Please see the `README` included in the `sj` distribution for version specific details, including known bugs and features to be completed soon. Questions and comments are very welcome: `raymond.hu05@imperial.ac.uk`.

2 Quick Start

The purpose of this section is to quickly get a simple **SJ** program compiled and running.

2.1 Installing SJ

The first thing to do is download a copy of the **SJ** distribution, containing the compiler and runtime libraries, from the **SJ** homepage [8]. Java 5 or later is required to both compile and run **SJ** programs.

After unpacking the archive, in the top `sj` directory you should see the following:

- `README` See this for details regarding the particular **SJ** version as well as general information.
- `bin` Contains scripts for running the compiler (`sjc`) and compiled programs (`sj`).
- `lib` Repository for the library classes needed to run the compiler and executables. The above scripts are set to automatically include this directory.
- `tests/src` Contains some example **SJ** programs, ready to be compiled and run.

As a test, entering

```
> bin/sjc
```

at `sj` should invoke the `sjc` compiler script, and return

```
sjc: No command line arguments given
usage: sjc [options] <source-file>.sj ...
where [options] includes:
  ...
```

because no source file was specified.

2.2 A Simple SJ Program

If we do, from `sj`,

```
> cat tests/src/quickstart/Alice.sj
```

we should see the **SJ** source file for the `quickstart.Alice` class, as in Listing 1. This is an example implementation of an Alice role, who will attempt a session with a Bob role (see below) in which she sends a greeting (a message of type `String`) and then waits for a reply. The comments in the first two lines give command line instructions for compiling and running this program.

Most of the actual code is just regular Java code: for now, we simply describe the key parts related to **SJ** session programming.

- SJ **package imports** (line 6). The libraries required by most typical **SJ** programs live in the imported packages: `sj.runtime.*` and `sj.runtime.net.*`.
- **Protocol specification** (line 11). The purpose of the Alice role is to interact with a Bob role. In **SJ**, the *protocol* by which the interaction should proceed is explicitly declared using *session types* [3, 1]. A session type describes the sequence and structure of communication actions to be performed from the perspective of a particular role. In this example, the `AliceToBob` protocol states that Alice should initiate the session (`begin`), then send the greeting (`!<String>`) and receive a reply (`?(String)`). See § 3.3 for a more comprehensive description of session types for protocol declarations.
- **Session server-addresses** (line 14). Alice is going to request a session from Bob: Alice is the client side, and Bob the server, of this binary session. To do so, Alice needs to know the address of Bob: this information is represented by a *session server-address* entity, an object of type `SJServerAddress`, which encapsulates the IP and TCP port of the server, and the type of the session offered by the server. Here, we

```
1  //> bin/sjc tests/src/quickstart/Alice.sj -d tests/classes/
2  //> bin/sj -cp tests/classes/ quickstart.Alice
3
4  package quickstart;
5
6  import sj.runtime.*;
7  import sj.runtime.net.*;
8
9  class Alice {
10   public static void main(String[] args) throws Exception {
11     protocol AliceToBob { begin.!<String>.?(String) }
12
13     SJServerAddress c =
14           SJServerAddress.create(AliceToBob, "localhost", 8888);
15     SJSocket s = SJFSocket.create(c);
16
17     try (s) {
18       s.request(); // begin
19       s.send("Hello Bob from Alice!"); // !<String>
20       System.out.println((String) s.receive()); // ?(String)
21     }
22     finally { }
23   }
24 }
```

Listing 1: `tests/src/quickstart/Alice.sj`

pass this information as arguments to the static `create` method of the `SJServerAddress` class.

- **Session sockets** (line 15). The communication actions that make up a session are performed via *session sockets*, objects of type `SJSocket`. Here, we create a new session socket by calling `create` on the `SJFSocket` class, passing as an argument the session server-address `c` to indicate that this socket is for conducting sessions with this particular server. `SJFSocket` and `SJRSocket` are the two concrete `SJSocket` implementations provided in the current version of SJ; the differences between them and how session sockets may be used are discussed in §3.4.

- **Session-try statements** (line 17). Session operations are performed within *session-try* statements. Session-try is an extension of the regular Java try that accepts session socket parameters: in this example, just `s`. Sessions may only be initiated within the scope of a session-try for which the corresponding socket is specified, and any initiated sessions must be completed, according to the associated protocol, within the scope of the session-try. The session-try in this example does not handle any of the potential exceptions itself, but catch blocks can be added in exactly the same way as the regular Java try. See §3.5 for more details about session-try.

- **Session operations** (lines 18–20). The communication actions comprising a session are performed much like method calls on the affiliated session socket object. In `Alice`, we use `request` to initiate the session with Bob, `send` to send the greeting message and `receive` to read Bob's reply. For this simple example, it is easy to see how this sequence of session operations corresponds to the `AliceToBob` protocol that we declared above (line 11). The **SJ** compiler statically verifies that session implementations conform to the intended protocols, thus guaranteeing the correctness of our communication behaviour. The session operations supported by **SJ** are presented in §3.6.

The code for Bob (Listing 2) comprises the server side of this session. The declared protocol (`BobToAlice`, line 11), and hence the subsequent session implementation, *duals* the corresponding protocol and session implementation in the above `Alice` class: basically, where Alice is sending a message, Bob is receiving, and vice versa. The main differences in the server side program are as follows.

```
1  //> bin/sjc tests/src/quickstart/Bob.sj -d tests/classes/
2  //> bin/sj -cp tests/classes/ quickstart.Bob &
3
4  package quickstart;
5
6  import sj.runtime.*;
7  import sj.runtime.net.*;
8
9  class Bob {
10   public static void main(String[] args) throws Exception {
11     protocol BobToAlice{ begin.?(String).!<String> }
12
13     SJServerSocket ss = null;
14
15     try {
16       ss = SJFServerSocket.create(BobToAlice, 8888);
17       SJSocket s = null;
18
19       try (s) {
20         s = ss.accept();
21         System.out.println((String) s.receive());
22         s.send("Hello Alice from Bob!");
23       }
24       finally { }
25     }
26     finally {
27       if (ss != null) ss.close();
28     }
29   }
30 }
```

Listing 2: `tests/src/quickstart/Bob.sj`

- **Session server-sockets** (line 16). Session server-sockets, objects of type `SJServerSocket`, wait for incoming session requests. We associate the intended protocol to the session server-socket when it is created, and state which local TCP port the server should listen on. In this example, we `create` a `SJFServerSocket` to match the `SJFSocket` of `Alice`.

- **Server accept** (line 20). The `accept` operation processes the next session request or blocks until one is available. If the request is for a compatible session, i.e. the session types of the two parties are indeed dual, then the `SJFServerSocket` of Bob returns an `SJFSocket` representing the server side endpoint for the newly established session that partners the session socket used by the client to request the session. The returned session socket `s` is used to implement the actions to be performed by Bob in this session (lines 20–22), according to the `BobToAlice` protocol. If the session request is not compatible, then the `accept` and `request` operations at each party raise a `SJIncompatibleSessionException` to abort the session.

Compiling `Alice.sj` and `Bob.sj`. The most convenient way to compile **SJ** programs is to use the `bin/sjc` script. From the top `sj` directory, we can separately compile each class using the commented commands in the first line of each source file (see Listings 1 and 2), or as a shortcut, we can just do:

```
> bin/sjc tests/src/quickstart/Alice.sj
      tests/src/quickstart/Bob.sj -d tests/classes/
```

Either way, the resulting Java source and binary files will be output to `tests/classes/quickstart`.

Running Alice and Bob. The `bin/sj` script is for running compiled **SJ** programs. In this example, Bob is hardcoded to open his session server-socket on TCP port `8888` (this example will fail if this port is not available), and Alice is similarly instructed to look for Bob on the `localhost` at that port. Being the server, Bob needs to be started first, otherwise Alice will fail to find Bob (since the port will not be open). From `sj`, we can do (as per the commented commands in the second line of each source file):

```
> bin/sj -cp tests/classes/ quickstart.Bob &
```

and then

```
> bin/sj -cp tests/classes/ quickstart.Alice
```

which will produce the output:

```
Hello Bob from Alice!
Hello Alice from Bob!
```

3 SJ Language Reference

The previous section illustrated the basic features of **SJ** through a simple example; this section presents a more detailed reference for session programming.

The current version of **SJ** is an extension of Java 1.4, i.e. generics are not supported; this is due to the third-party tools used in the current implementation of the compiler (see §4).

3.1 SJ Background and Motivation

The purpose of **SJ** is to provide direct language support for (binary) *session programming* as an extension to Java. As illustrated in the preceding section, session programming starts from the description of protocols for interaction (using session types), which can then be concretely implemented using a set of structured communication operations available on session sockets. These operations provide the building blocks for interactive computing in an analogous way to how standard (imperative) language constructs are used to build sequential algorithms. The **SJ** compiler statically verifies that session implementations conform to the protocol specifications; combined with a runtime compatibility validation between the peers at session initiation, **SJ** programs guarantee communication safety.

Session programming in **SJ** has several benefits in comparison to widely-used alternatives such as regular socket programming and Java RMI. Byte stream communication through raw network (TCP) sockets, e.g. `java.net.Socket`, is inherently untyped, and the intended communication protocols have no direct representation in the program, neither as types nor programming constructs. These limitations can make communications programs more difficult to read and write, and also to verify. RMI and other RPC-based technologies provide a type-safe procedure (method) call abstraction to distributed objects. The fixed call-return shape of procedure call is, however, not suited to expressing certain interaction patterns, and there are no natural means of treating a sequence of such calls as a one complete unit of interaction, i.e. a session.

3.2 Overview of SJ Session Programming

First, we must understand the terms *session* and *session type*.

- Session programming is for systems and applications where the constituent parties or components interact according to certain protocols: session types are formal specifications of such protocols. Session types describe structured sequences of interactions including basic message passing, branching and repetition.
- A session is an instance of a session type, i.e. the unit of interaction encapsulating one run of a protocol. From the perspective of the abstraction, each session, including those of the same type, is conducted without interference on a separate channel.

Session programming in **SJ** comprises the following stages.

1. The first step is to *design the protocols* by which the parties should interact, and specify them as session types.
2. The above protocols are explicitly incorporated into the programs for each party or component: we *declare the session types* for the interactions to be performed.
3. The actual interactions that comprise a session are implemented using the *session programming constructs and operations*. The session operations are performed like method calls on session socket objects, endpoint handles to the session channels.
4. Each session implementation is *statically verified* by the compiler against the associated protocol according to the type system for session programming. Session implementations must respect the property of *session linearity* in terms of control flow and object aliasing.

The following sections describe the core features of **SJ** and demonstrate their use across the stages of session programming.

3.3 Protocol Declaration

Session programming begins by declaring the protocol for the intended interaction as follows,

```
protocol name { ... }
```

```
T   ::=  T.T                          // Sequencing.
       | begin                       // Session initiation.
       | !<M>                        // Message send.
       | ?(M)                        // Message receive.
       | ⊕{L1:T1, ...,Ln:Tn}         // Session branching.
       | ⊕[T]*                       // Session iteration.
       | rec L[T]                    // Session recursion scope.
       | #L                          // Recursive jump.
       | @p                          // Protocol reference.

M   ::=  Object type | Primitive type | T   // Message types.
L   ::=  Label
⊕   ::=  ! | ?
```

Figure 1: Session type syntax.

where `name` identifies the protocol, following the standard Java naming rules. Protocols may be declared as both local and field variables, although field protocols are currently restricted to `private` access.

The body of the protocol (inside the curly braces) is a *session type*, given by the syntax rules in Figure 1. The session type specifies how a session should proceed in terms of the actions that the particular party should perform, i.e. a view of the session from the perspective of that party. The key point is that the implementation of a session (see §3.6) is governed by the associated protocol: the **SJ** compiler statically verifies session implementations against the protocols.

The kind of actions that can be specified include basic message passing, conditional and repeated behaviours. The session type elements that describe these actions are explained below in pairs according to the *duality* relation between corresponding elements. Session type duality ensures that two parties implement compatible protocols, checked through a runtime validation at session initiation. Duality essentially means when one session party is sending a message, the peer is (or will be) expecting a message of that type (or supertype), thus precluding dynamic type errors from message communication.

Interaction sequences. If T and T' are session types, then

$$T.T'$$

is the session type that describes first performing the interactions of T followed by the interactions of T'.

Session initiation. The action of initiating a new session with a peer is denoted by:

`begin`

Unlike most of the other session type constructors, `begin` is symmetric, meaning that both parties have the same specification of this action[1]. Note that `begin` can only appear as the initial prefix at the top level of a session type, since a session can only be initiated once. **N.B.** there is no explicit session type for the action of closing a session: the end of a session is implicit at the end of a type.

Basic message passing. Sending and receiving a message of a type M are respectively described by the dual types

`!<`M`>` `?(`M`)`

Message types are serializable Java object types, primitive types and session types (the latter are explained in 'Higher-order Message Types' below). The notion of duality between send and receive includes *session subtyping* [2], which allows a message subtype to be sent where a supertype is expected. In general, the '`!`' ('`?`') symbol in session types denotes some form of output (input) action, of which send (receive) is a particular instance.

Branching. Sessions can branch into one of multiple paths: one of the parties is responsible for making the branch decision (which path to select), and the other party must follow. Paths are identified by string labels.

`!{`L_1`:`T_1`, ..., `L_n`:`T_n`}` `?{`L_1`:`$\bar{T}_1$`, ..., `L_n`:`$\bar{T}_n$`}`

The type on the left specifies that the session party is the decision maker and may select one of the paths labelled from `L`$_1$ to `L`$_n$. The dual type on the right (where $\bar{T}_x$ is a dual type to T_x.) says the session peer should follow the first party's decision and take the corresponding path. Session subtyping allows the select type to select from a subset of the paths that the corresponding branch type supports, i.e. from labels L_m, where m is a subset of n.

Note that the type,

[1] This may be revised in a future version of **SJ** to distinguish the client and server sides.

$$\oplus\{L_1{:}T_1,\ \ldots,\ L_n{:}T_n\}.T'$$

means that, whichever branch is selected, the session will then proceed according to the type T' (if and) after the branch has been completed.

Iteration. Sessions may involve the iteration of some subsession. Similarly to branching, one session party is responsible for making the decision at each iteration whether or not to iterate, and the session peer must follow. The syntax for the dual iteration types is

`![`T`]*` `?[`$\bar{T}$`]*`

where $\bar{T}$ is a dual type to T. The decision to iterate or not is made before each iteration, so the subsession may be performed zero or more times.

Recursion. Repeated behaviour in session types can also be specified using recursion, which has the type syntax

`rec` L`[`T`]` `#`L may occur in T

where L is an alphanumeric label, and '`#`' is just a symbol which essentially means "jump" back to the protocol state at which the label was declared (`rec` L) and repeat T (although different paths can of course be taken through T each time round). Recursions with differing labels can be nested: `#` can jump back out to any nesting level by specifying the appropriate label. **N.B.** only *tail* recursion is supported, so the corresponding syntactic restrictions apply on where `#`L can occur in T.

Note that recursion uses a symmetric type constructor. The decision whether or not to repeat the subsession is not directly associated to the recursion itself, but rather depends on a branch nested within the recursion: generally some paths should lead to a `#`L and some should not. However, recursions that never "quit" (except on failure) may be useful for certain situations.

Higher-order message types. As mentioned above, message types can themselves be session types. Whereas the "regular" message types specify the communication over some established session, the *higher-order* communication types additionally express changes in the shape of the session network. For example, the dual types

`!<?(int)>` `?(?(int))`

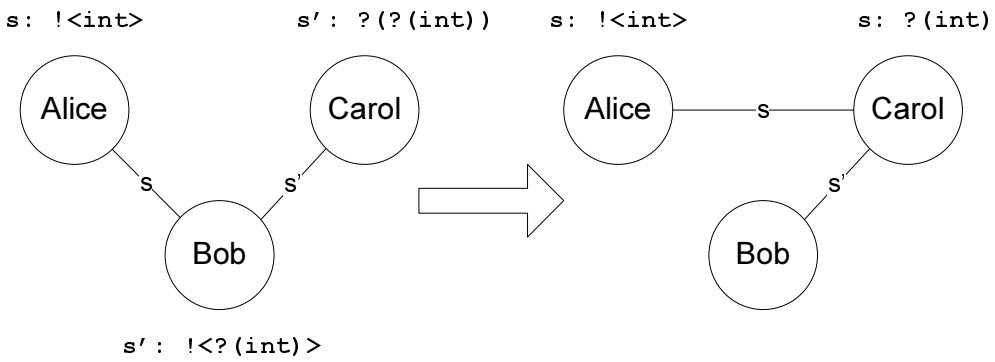

Figure 2: Delegation of `s` from Bob to Carol.

respectively say that we should send and receive a *session* of type `?(int)`. Higher order session communication is often referred to as session *delegation*.

Figure 2 illustates a basic delegation scenario. The left depicts the session configuration before the delegation is performed: Alice is engaged in a session `s` of type `!<int>` with Bob, and Bob is also in a session `s'` of type `!<?(int)>` with Carol. Then instead of accepting the integer from Alice himself, Bob can *delegate* his role in `s` to Carol so that she will receive this message. This delegation action corresponds to Bob's higher-order send type for the session `s'` with Carol.

The right hand side of Figure 2 shows the change in session configuration after the delegation has been completed: Alice is now interacting with Carol for the session `s`. The key point is that by accepting the delegated `s`, Carol makes an implicit promise to finish the session on behalf of Bob; thus, Carol will perform `?(int)` to match the action of Alice. Because Bob has delegated full responsibility for completing `s` to Carol, he will proceed as if his contract for this session has been fulfilled.

An important property of delegation is that the type of the session being delegated does not itself convey any information about the delegation: the delegation action is between Bob and Carol, and is transparent to `Alice`, the passive party. Because Carol will fulfil the session contract originally signed by `Bob`, i.e. as Alice expects, session delegation respects communication safety.

Another useful application of higher-order session types is the communication of types like

`!<begin.`T`>` `?(begin.`T`)`

which do *not* specify the delegation of an active session (as in the above example); since the nested session type is prefixed by `begin`, the session has not yet been initiated. Rather, this communication informs the recipient about another peer from which new sessions of the specified type may be requested. In other words, this permits the creation of new edges in a session configuration as well as the migration of existing edges. This type of action arises naturally in many real world settings, such as FTP.

Higher-order communication is a key feature of session programming, enabling type-safe programming for mutually-transparent endpoint mobility.

Referencing other protocols. For syntactic convenience, one protocol can be referenced from another using the `@` operator. For example, given

```
protocol p1 { ... }
```

we can write protocols like

```
protocol p2 { ... .@p1 ... }
protocol p3 { ... .!<@p1> ... }
```

with the expected meaning: the `@p` is syntactically substituted for the protocol of that name. **N.B.** protocols cannot make forward references nor reference themselves: (mutually) recursive protocol declarations are not permitted. This is in keeping with the standard Java rules for variable reference, including fields.

3.4 Session Sockets

After declaring the required protocols, we create *session sockets* for implementing the actual session code. Session sockets represent the endpoints of a session connection: each of the two peers owns one endpoint and performs the specified interactions via the **SJ** session operations on that endpoint.

Session sockets are objects that extend the abstract `SJSocket` class. The current version of **SJ** provides two types of session socket, `SJFSocket` and `SJRSocket`, both of which employ TCP as the underlying transport; **N.B.** the two socket types are not currently interoperable. The differences between the two are mostly related to the mechanisms for handling session delegation: the first implements the *bounded-forwarding* protocol and the second implements the *resending* protocol, as presented in [4]. However, these details (such as impact on performance [8]) are not important for most users, only that the two socket types are not yet compatible.

As in many regular socket programming libraries (e.g. `java.net.*`), we distinguish session *client* and *server* sockets. The former are used to request sessions from the latter.

3.4.1 Session Client Sockets

Session client sockets (or just session sockets for short) are created by calling the static `create` method on the desired socket class, `SJFSocket` or `SJRSocket`. This method takes as a parameter an object of type `SJServerAddress`, which binds the IP address and TCP port of the target session server with the type of the session supported by that server. `SJServerAddress` objects are also created by calling a static `create` method on the class, passing as parameters the information just described.

For example, assuming we have a protocol `p`,

```
SJServerAddress c = SJServerAddress.create(p, "host", 1234);
SJSocket s = SJFSocket.create(c); // Or use SJRSocket.
```

first creates the *session server-address* `c`, associated with the protocol (session type) `p`. `c` is then used to create the session socket `s` (of type `SJFSocket`), which means that `s` should be used for sessions of type `p` with the specified server, in this case located at port `1234` on the machine identified by `"host"`.

An important requirement in **SJ** is that the session-typed objects, such as `SJSocket` and `SJServerAddress`, are not permitted to be aliased. In particular, assignment both from and to session-typed variables is forbidden, as is passing the same session-typed variable as multiple arguments in the same method call (see §3.7). The only exception is the assignment of a freshly created session-typed object to a null-initialised and fresh (that is, not previously assigned to) variable, e.g.

```
SJSocket s = null;
... // 's' is not assigned to yet.
s = SJFSocket.create(c);
```

Session client sockets cannot be "reused", i.e. once a session has been completed, it cannot be run again using the same session socket object.

3.4.2 Session Server-sockets

Session server-sockets are the counterpart to the above session (client) sockets. Session server-sockets are created in much the same way,

```
SJServerSocket ss =
     SJFServerSocket.create(q, 1234); // 'q' is a protocol.
```

but are created *active*, meaning that the specified port is opened and `ss` is immediately ready to accept session requests. Unlike server-address objects and client sockets, creating a server-socket can throw a `SJIOException`, typically if there is a problem opening the local port. If this is the server being targeted by the client socket created above, then the protocol `q` should be of dual type to `p`.

3.5 Session-try Statements

Sessions are implemented within *session-try* statements. Session-try is very similar to the regular Java try, but also takes as parameters the session sockets for sessions that may be implemented within its scope. Furthermore, any sessions initiated within a session-try must be completed within the scope of that session-try. Active sessions are automatically closed when leaving the scope of a session-try.

```
try (s1, s2) { // Assume session sockets 's1' and 's2'.
  ... // Implementation of 's1' and/or 's2' sessions.
} catch (...) {
  ...
} finally {
  ...
}
```

As for regular try statements, the `finally` clause is optional if there is a `catch` clause, and vice versa. The implementations of `s1` and `s2` may be interleaved along with other Java code, provided that they respect their protocols and the other rules of the session type system.

The **SJ** compiler statically verifies that initiated sessions are completed within the parent session-try according to the relevant protocols. However, the completion of a session includes the session being delegated (§3.3), passed as an argument to a **SJ** session method (§3.7) or handed over to a **SJ** session thread (§3.8); each of these actions implicitly complete the sessions involved with respect to the current session-try scope.

Session-try statements can be nested, although sessions specified as a parameters to an outer scope cannot be accessed from within an inner scope, except if the outer session is being delegated over an inner session. In such a case, no further actions need be (nor can be) performed for the delegated session in its parent scope. The programmer is free to decide whether exceptions raised in an inner session-try are consumed there, or thrown out to the outer scope; this depends whether the failure of the nested session should fail the outer session.

```
s.request();                                          // begin
s.send(m);                                            // !<M>
s.receive();                                          // ?(M)
s.outbranch(L) {P}                                    // !{L:T}
s.inbranch() {case L1:{P1}... case Ln:{Pn}}           // ?{L1:T1, ..., Ln:Tn}
s.outwhile(cond) {P}                                  // ![T]*
s.inwhile() {P}                                       // ?[T]*
s.recursion(L) {P}                                    // rec L[T]
s.recurse(L);                                         // #L
```

Figure 3: The **SJ** session operations and their types.

3.6 Session Operations

After creating a session socket, a session can be implemented within a session-try using the *session operations*. The following sections describe the main session operations, listed in Figure 3, for performing the interactions specified by the associated protocol declaration §3.3. The session operations are generally invoked via session sockets in a method call-like manner. As stated in the preceding section, session implementations must be completed within the scope of the parent session-try statement.

3.6.1 Session Initiation

The `begin` type is implemented by the session socket `request` and server-socket `accept` operations. The server `accept` operation blocks until a `request` is received: if the two parties support compatible session types, then `accept` returns a session socket as the server-side endpoint of the new session, matching the client side socket used to make the session request. Otherwise, a `SJIncomaptibleSessionException` is raised at both parties to abort the session.

```
SJSocket s = ss.accept();                    s.request();
```

Compatible means that the session peers implement *dual* protocols, meaning that when one sends a message, the other will expect a message of that (super) type (correspondence between `!` and `?` actions), and allowing for branch subtyping (one can select from a subset of the choices offered by the other).

Note that each type of session socket (`SJFSocket` or `SJRSocket`) can currently only be used to initiate sessions with the corresponding server-socket type.

3.6.2 Basic Message Passing

Message passing in **SJ** is asynchronous but lossless and order preserving, meaning that send operations complete immediately, but receive operations block until a message is available, and that all messages sent over a particular session will be received by the session peer in the same order they were sent.

Serializable objects (implements `java.io.Serializable`) and primitive types are sent using the `send` operation. (Non-serializable objects cannot be communicated.)

```
s.send(myObject);          // !<MyObject>
s.send(123);               // !<int>
```

For code clarity, dedicated `sendBoolean`, `sendInt`, etc., for the primitive types are also supported. However, dedicated variants *must* be used to receive primitive types.

```
String m = s.receive();          // ?(String)
int v = s.receiveInt();          // ?(int)
```

No cast is needed for `receive` operations, as this information is inferred from the expected type according to the protocol being implemented. **N.B** the type of the `?(String)` receive in the above example is being inferred in this way, and *not* from the assignment of the received object to the `String` variable. Explicit casts are, of course, still permitted.

3.6.3 Branching

The `outbranch` and `inbranch` operations respectively select and switch on a branch label.

```
// !{LAB:...}
s.outbranch(LAB) {
  ...
}
```

```
// ?{LAB..., ...}
s.inbranch() {
  case LAB: {
    ...
  }
  ...
}
```

Operationally, the `outbranch` party implicitly sends the decision to the `inbranch` peer so that both select the same path. One can view session branching as something like a distributed or interactive version of the regular Java switch statement.

3.6.4 Iteration

Session iteration is performed using the `outwhile` and `inwhile` operations. The argument to `outwhile` is a boolean expression; `inwhile` takes no arguments.

```
// ![...]*
s.outwhile(...) {
  ...
}
```

```
// ?[...]*
s.inwhile() {
  ...
}
```

The result from evaluating the boolean condition at each iteration is implicitly sent by the `outwhile` party to the `inwhile` peer, so that both parties iterate and exit in tandem. `outwhile` and `inwhile` are like interactive versions of regular iteration constructs, such as the `while` statement

3.6.5 Recursion

The session `recursion` construct and `recurse` operation take a label as arguments. `recursion` marks the scope for a recursive subsession under the specified label, and `recurse` makes the recursive "jump" back to the start of the specified scope.

```
// rec L[...]
s.recursion(L) {
  ...
}
```

```
// #L
s.recurse(L);
```

N.B. Session recursion is restricted to tail recursion: neither session operations nor regular Java code may follow a `recurse` at any lexical scope level up to the outermost recursion scope that may be jumped to. However, `recurse` can jump to any valid outer recursion scope, i.e. not just the innermost recursion scope. Unbound `recurse` labels are not permitted. Note that neither `recursion` nor `recurse` perform any actual communication.

3.6.6 Higher-order Communication

To delegate a session, we simply pass the session socket variable for the session being delegated as an argument to a `send` operation on the target session.

```
s1.send(s2);     // !<T>, where T is the remaining
                 // session type of 's2'.
```

Only active session sockets can be delegated, and delegation implicitly completes the delegated session; in the above example, no further operations are permitted on `s2`. Note that it is not safe to delegate an outer session inside an iterative or recursive context. Such use would be detected as error by the compiler.

The `receive` operation receives delegated sessions.

```
SJSocket s2 = s1.receive();   // ?(T), where T is the expected
                              // session type for 's2'.
```

Casts are optional, as for ordinary receive operations.

```
s2 = (T1)s1.receive();     // ?(T1)
s3 = (@p)s1.receive();     // ?(T2), where T2 is the session
                           // type declared by 'p'.
```

The communication of session server-addresses is similar.

```
s.send(c);                                  // !<begin....>
SJServerAddress c = s.receive();            // ?(begin....)
```

Like the operations for communicating primitives, dedicated `sendSession` and `receiveSession` can be used for clarity; similarly for server-addresses.

3.6.7 Combining Session Operations and Regular Java

Regular Java also plays a part in session implementation. For instance, if-statements are naturally treated as conditional session contexts: sessions can be nested within a branch, and operations on outer sessions must be consistent across all branches. For example, whereas

```
if (...) s.send(...);
```

is simply illegal,

```
if (...) {
  s.send(...);
} else {
  s.send(...);
}
```

is fine, provided that the `send` operations do indeed conform to the expected session type. A slightly more complicated version of this is, for example,

```
// !{ LAB1:..., LAB2:... }
if(...) {
  s.outbranch(LAB1) { // !{ LAB1:... }
    ...
  }
} else {
  s.outbranch(LAB2) { // !{ LAB2:... }
    ...
  }
}
```

which allows us to select different branches based on the condition of the if-statement.

While-loops, etc. are similarly treated like iterative session contexts: complete sessions can be nested within a loop, but operations on outer sessions is not type safe. For example,

```
while (...) {
  s.send(...);
}
```

is illegal, but a common, valid pattern is

```
while(...) {
  try (s) {
    s = ss.accept(); // 'ss' is a server-socket.
    ... // Implementation of 's'.
  }
  catch(...) {
    ... // Don't bring down server just because of one bad client.
  }
  ...
}
```

which handles multiple (concurrent) session requests made to the server-socket.

3.7 Session Methods

Session sockets can be passed as arguments to, and returned from, *session methods*. Session methods are the same as regular methods, except the parameter or return type for session values should the expected *session type* (not `SJSocket`). The current version of **SJ** restricts session methods to `private` access.

The following is an example session method that takes a session type argument as well as an integer.

```
  try (s1) {
    ... // Partial implementation of 's1'.
    finishSession(s1, 123); // 's1' remaining session type 'T1'.
  }

...

private void finishSession(T1 sarg, int i) throws SJIOException {
  ... // Finish 'sarg' according to 'T1'.
}
```

The key point is that the type of the session argument `sarg`, `T`, must correspond to the remainder of the session `s1` at the point of the method call. Session methods serve as implicit session-try scopes for the session arguments. Hence, the implementation of `sarg` can be continued directly within the method body. By the standard rules for exceptions, any exceptions that may be raised by the session operations used, such as `SJIOException` must be thrown out by the method back to the parent session-try.

The next example, listed in Figure 4, returns a session from a session method. The session `s2` is initiated and partially implemented within `getSession` before it is returned to the calling context, where the remainder of the session is implemented. Exceptions that may be raised by the session operations within the `s2` session-try must be thrown out to the calling context. This is enforced by the fact that the `s2` session-try cannot catch any exceptions itself, as this would not guarantee that a session of type `T2` is returned; the compiler rejects such cases. Inactive session sockets, i.e. uninitiated sessions of `begin`... type, cannot be returned.

3.8 Session Threads (Under Development)

A session socket can be passed to a *session thread* using the `spawn` operation, meaning that the (remainder of the) session is to be performed by that

```
  try (sret) {
    ... // 'sret' is null-initialised and not yet assigned to.
    sret = getSession(123);
    ... // Finish 'sret' according to 'T2'.
  }

...

private T2 getSession(int i) throws ... { // Multiple exceptions.
  ... // 's2' created.
  try (s2) {
    ... // 's2' partially implemented.
    return s2; // 's2' has remaining type 'T2'.
  }
  finally {
    ...
  }
}
```

Figure 4: Returning a session from a session method.

thread. This operation is in effect a "local" delegation, that is, a delegation between parties in the same JVM instance. **N.B.** Session threads are currently under development; in particular, exception handling for session threads is incomplete.

Following is an example session thread class.

```
class MyThread extends SJThread {
  public void run(T1 s1, T2 s2) throws SJIOException {
    ... // Implementation of 's1' and 's2'.
  }
}
```

Session threads extend the `sj.runtime.net.SJThread` class. Like `java.lang.Thread`, `SJThread` must declare a single `public void run` method containing the thread body. The differences are that a session thread `run` accepts arguments including session values, the method can throw exceptions, and the thread is started immediately upon creation. **N.B.** Currently, session thread exceptions serve only to terminate the thread and cannot be directly handled.

Like other session methods, `run` serves as an implicit session-try scope for its session arguments. The only difference is that uninitiated sessions (those of type `begin...`) are automatically incorporated into the try-scope of session thread `run` methods.

Given the above session thread class, the `spawn` operation can be used as follows.

```
... // 's1' and 's2' created and partially implemented.
<s1, s2>.spawn(new MyThread()); // Note the '<...>' syntax.
```

`spawn` takes as an argument the session thread instance to which the target session should be passed. However, `spawn` differs from the other session operations in that it can operate on multiple sessions, passing them all to the specified thread. The types of the sessions being passed, `s1` and `s2`, must match the parameters of the `run` method declared for the session thread class, `MyThread`, at the point of the `spawn`.

3.9 Eager Remote Class Loading and Verification

Due to session subtyping, in particular message subtyping, it is possible for a session peer to lack some of the classes needed to deserialize an object received from its peer. For this purpose, the **SJ** session sockets support a remote class downloading feature similar to that used by Java RMI.

```
// 's' is an inactive session socket.
s.setCodebase("http://www.doc.ic.ac.uk/~me/codebase/");
s.setRemoteClassLoading(true);
```

Session sockets can be configured to look for missing classes at a codebase, a HTTP class repository. After creating a session socket, but before the session is initiated, we can set the codebase parameter on the socket and enable remote class loading. If the required class cannot be retrieved from the codebase, then the usual `ClassNotFoundException` is raised. The class loading parameters cannot be modified after the session has been initiated.

Because session types describe all the messages that may be communicated in a session, the session type of the peer, obtained at session initiation can be used to conservatively anticipate all potentially missing classes. These classes can then be *eagerly* downloaded before (and if) they are actually required. Similarly, compatibility between the classes that both parties share can be eagerly verified.

```
// 's' is an inactive session socket.
s.setEagerClassDownloading(true);
s.setEagerClassVerification(true);
```

Note, however, that eager class downloading does not guarantee to load *all* potentially needed classes, only those for all specified message types; it is possible to send a subtype of the specified message.

4 About the SJ Implementation

The full distribution of **SJ** includes the source code for the compiler and runtime libraries [8] under the `compiler/src` and `runtime/src` directories respectively. The compiler is implemented as an extension to the Polyglot compiler framework (version 2.2.3) [6], and the project can be built using Ant [5] (a compiler for Java 5 or later is required). The default (empty) Ant target will place the resulting classes in `compiler/classes` and `runtime/classes` accordingly; the `jar` target will compile the source and make `sj.jar` and `sj-rt.jar` in the `lib` directory. The `sjc` and `sj` scripts will first look for dependencies in the `classes` directories; if they cannot be found, then the scripts will look for the `jar` archives in `lib`. Compiling and running **SJ** programs require both the compiler and runtime class files.

References

[1] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida and S. Drossopoulou, *Session Types for Object-Oriented Languages.*

[2] S. Gay and M. Hole, *Subtyping for Session Types in the Pi Calculus.*

[3] K. Honda, V. T. Vasconcelos, and M. Kubo., *Language Primitives and Type Disciplines for Structured Communication-based Programming.*

[4] R. Hu, N. Yoshida and K. Honda, *Session-based Distributed Programming in Java.*

[5] Apache Ant, `http://ant.apache.org/`.

[6] Polyglot extensible compiler framework, `http://www.cs.cornell.edu/projects/polyglot/`.

[7] **SJ** Language Specification (draft).

[8] The **SJ** project homepage, `http://www.doc.ic.ac.uk/~rh105/sessionj.html`.